THE BELL SYSTEM TECHNICAL JOURNAL
SAFEGUARD SUPPLEMENT
Printed in U.S.A.

SAFEGUARD Data-Processing System:

Systems Programming in PL/1

By P. A. VAN SCIVER

(Manuscript received January 3, 1975)

This paper deals with the development of a large systems program in a high-level language. The reasons for selecting a high-level language, the most extensively used features, the benefits derived, and the significant problems encountered are described.

I. INTRODUCTION

This paper highlights the important aspects of developing a large systems program in a high-level language. The Execution Preparation Facility (XPF) performs the linkage editor function on the SAFEGUARD project. When XPF was originally designed, the decision was made to develop it in PL/1. The paper examines the most extensively used features of PL/1, describes the problems encountered during development, points out the lessons learned, and discusses the benefits derived from the use of a high-level language. An appendix provides XPF development productivity data and comparisons.

II. FUNCTIONAL DESCRIPTION

XPF is the last major step through which software must pass on its way to execution on the CLC. Some functions performed by XPF can be compared to those of the operating-system linkage editor in that XPF prepares the output of the language processor for execution, sets up the overlay environment, and produces memory maps and cross-reference listings.

The output of XPF, called a thread, is a collection of programs and data sets and their associated control tables bound to absolute addresses. The thread also contains installation, debugging, and data reduction information. Inputs to XPF are user-supplied commands, execution time parameters, assembler or compiler output, a partitioned data set called the system file that describes the CLC operating system, and, in an update mode, the results of previous XPF runs.

The major functions of XPF are construction of control vector tables (CVTs) for interthread linkage, allocation of CLC resources, primary memory and disc storage, binding of thread units, and construction of operating-system control tables (PCTs). In addition, XPF produces a series of printed listings describing memory configuration, process structure, forward and back referencing among units, PCT construction, and thread summary data.

III. PHYSICAL DESCRIPTION

XPF consists of 246 subroutines, 95 percent of which are written in PL/1. The internal structure is modular. Functions are performed by 24 independent modules that overlay each other. The XPF load module consists of 130 overlay segments and requires 2.5 megabytes of disc storage. The access method used to retrieve object code, while not actually a part of XPF, is also included in the XPF load module.

XPF operates in a 400-K region, of which 260 K is occupied by the overlaid load module. During execution, 12 internal files are used for work space and intermodule communication. Since the disc space needed for these files varies with the input, space allocation is controlled by catalogued procedure parameters. The actual execution of XPF is controlled by the execution time parameter field on the user's JCL execute card. Most modules execute at the option of the user and are controlled through this field. The mode of execution (regular, debug, or update) is also directed by execution time parameters.

IV. DESIGN DECISIONS

Since most systems software is written in assembly language, one question arises: Why was a high-level language used for this facility? Three major factors contributed to this decision.

(*i*) Development time was short. It was felt that the anticipated ease of writing in a high-level language, coupled with extensive utilization of compiler-provided debugging capabilities, would help provide the desired results within the allocated time. This proved to be the case, and each of ten XPF releases was produced on schedule.

(*ii*) A high degree of flexibility was required. XPF, the operating systems, and the applications processes were to be developed concurrently. Since XPF is the software that links the operating systems and the applications processes, responsiveness to the design requirements of both groups was a necessity. A high-level language was judged to be best equipped to provide the required flexibility. This approach proved valid. In practice,

when a SAFEGUARD design problem could have been solved by changing the CLC operating system, the applications processes, or the XPF, XPF was usually chosen.

(*iii*) The execution of XPF was expected to be I/O limited. Therefore, potential compiler-generated CPU inefficiency was not a major consideration.

Since XPF was to be developed and executed under OS/360, the contenders for a high-level language were PL/1, FORTRAN, COBOL, and ALGOL. PL/1 was an easy choice. The bit-handling capabilities of the language were well known, and many members of the development group had PL/1 experience.

V. HOW PL/1 WAS USED

This section records those features of PL/1 used most extensively in the development of XPF.

External variables were used to store relatively small amounts of data needed throughout XPF execution. Since the external variables were located primarily in the root segment of the load module, their use in intermodule communication aided in segmentation and structuring of the overlay tree.

Static storage was used extensively to take advantage of what would have been dead space in the short legs of the overlay tree. The judicious use of static storage minimized the amount of memory required for execution. Static variables require special attention in an overlay environment. Every time a segment is brought into memory, each static variable is reinitialized, but in subsequent calls to the segment that do not require overlay, the variables retain their current values.

Three types of I/O were utilized. Stream-oriented I/O was used for printed listings and debugging output. Sequential-record-oriented I/O was used for intermediate files for communicating between, at most, two modules. The TITLE option was used with these files to allow many modules to utilize the same disc area, thereby reducing overall resource requirements. Regional I update files were used to satisfy global communication requirements, e.g., paging of data and storage of object and bound units.

Area variables were utilized by many modules. Each record entered into the update files consisted of a single area variable. Individual data entries were allocated within the area and entry addresses assigned. The use of areas avoided excessive I/O by allowing large amounts of data to be stored on a single record. The utilization of PL/1 area management greatly reduced the amount of user-supplied code necessary for record formatting and control mechanisms.

In the shorter legs of the overlay tree, area variables declared with the static attribute were employed, realizing the advantages described earlier.

Based variables were used extensively, especially in the areas of I/O. Based structures were declared in the calling programs, and file managers returned pointers to the requested items.

List processing was a major requirement in XPF design. Frequent sorts of these lists were required. The use of linked lists prevented excessive data movement during sorts, since only the pointers needed to be modified to change the order of the tables.

The bit handling features of PL/1, an important aspect of the decision to use this language, were used extensively. Since the CLC uses ASCII character representation, characters had to be interpreted as bit strings.

Label arrays were utilized in command processing. Since many commands contained common keywords and fields, processing was broken down to that level. Keywords and fields were interpreted and assigned number values that were used as indices into label arrays for keyword processing.

The PL/1 preprocessor played an important role in XPF development. Preprocessor statements and procedures were placed on a file that was accessed via "% INCLUDE" by all procedures. Four key functions were performed by the preprocessor:

(*i*) Declarations of global data such as external variables and based variables used in I/O were stored on the file and brought into each procedure that utilized them. This assured identical variable declarations throughout XPF.

(*ii*) Declarations of utility and file manager entry points and their associated parameter attributes were also placed on the file. This helped assure the consistency of parameters passed to these subroutines.

(*iii*) Certain constants such as area sizes, array dimensions, and conversion constants were subject to frequent change while optimal values were being ascertained. Programs referencing these constants did so via preprocessor variables. When modifications were necessary, the values of the preprocessor variables were changed on the file and the referencing programs were recompiled.

(*iv*) Preprocessor procedures were provided for frequently used in-line code.

VI. HOW ASSEMBLY LANGUAGE WAS USED

While 5 percent of the subroutines in XPF are written in assembly language, these amount to less than 0.2 percent of the total number

of machine instructions. Assembly language subroutines fell into two categories: data conversion subroutines originally written in PL/1 and recoded in assembly language for reasons of storage economy or efficiency, and subroutines written in assembly language to provide facilities not directly available in PL/1.

An example of the first is a TRANSLATE function that converts ASCII to EBCDIC, and vice versa. This function was not supported in PL/1 Version 4. By recoding in assembly language, a 20-K-byte subroutine was reduced to 500 bytes and made much faster.

An example of the second is a routine to access a partitioned data set of twenty or so members. Had this routine been written in PL/1, one DD card for each member of the data set would have been required.

VII. MAJOR PROBLEMS ENCOUNTERED

The most serious problem encountered during development was an obscure but critical bug in object code generated by PL/1 Version 4 that became important when a new computer with a larger memory was installed. XPF would ABEND if loaded in the upper third of memory because of bad code generated for bit-string operations. This made it necessary to convert XPF to Version 5 of PL/1. Incompatibility between these versions required complete recompilation and some recoding. Six weeks of effort were required to complete the conversion.

Another major problem was directly related to this conversion. Half-word storage, implemented in Version 5, caused structure alignments to be altered. Since boundary alignments were not required on the development computer, some problems were not detected. It was later discovered that XPF would not work on certain models of the IBM System 360. The most expedient method of correcting the problem was to declare the offending structures unaligned. Portability of XPF could probably have been ensured in advance by constantly being aware of the consequences of PL/1 defaults.

The XPF execution problem causing the most impact was excessive I/O usage generated by the OS overlay manager and not by PL/1. Dramatic reduction in load module accesses was accomplished by overlay restructuring.

VIII. LESSONS LEARNED

In addition to the initial decision to use PL/1, throughout the development of XPF many design and implementation decisions concerning the use of PL/1 were made. Some of these proved to be sound, and others had unfortunate results. This section deals with the results of these decisions.

The extensive use of the PL/1 preprocessor proved to be an excellent control mechanism. The inclusion of macros, entry point declarations,

and global variable declarations via preprocessor procedures greatly facilitated intermodule communication. This standardization guaranteed the integrity of interfaces.

As originally expected, the liberal use of PL/1 debugging aids was an invaluable development tool. The large number of logic errors detected through ON conditions such as SUBSCRIPTRANGE and STRINGRANGE underlines the value of their use.

PL/1 provides no debugging aids for pointer variables, used extensively in XPF, so it was frequently necessary to examine a dump to ascertain the exact nature of a problem. Since no error control philosophy within XPF had been established, dumps could not be produced at will. A global error control mechanism was instituted. By placing a single ON ERROR block in the main procedure and removing them from lower-level subroutines, the problem of inappropriate or inadequate response by these subroutines was eliminated.

No global coding conventions were established at the beginning of the project. This resulted in various methods of implementation of the same basic requirements, some of which were more efficient than others. A subset of PL/1 should be extracted that is most efficient for the particular application. Programmers should be warned to avoid certain implementation methods and encouraged to use other more efficient ones.

Since XPF was required to execute in a 400-K region (the maximum size for an express run), the use of small independent subroutines that could be overlaid was encouraged. In the longer legs of the overlay tree, this philosophy proved valid. However, in the shorter legs of the tree, this introduced unnecessary inefficiencies because of operating system overhead. The increased use of static storage in the shorter legs decreased the effect, but the use of fewer subroutines would have been more efficient.

The use of assembly language subroutines, though dictated by reasons of efficiency and necessity, presents some disadvantages. Since parameter definition is compiler-dependent, assembly language subroutines must be coded to meet the parameter passing standards and conventions of a specific compiler. In PL/1 these proved even more limiting since assembly language subroutines must be coded for a specific version of the compiler. When such subroutines are utilized, this dependency on a particular version of a compiler should be explicitly documented.

The assembly language complications are the most obvious of the compiler dependency problems. However, as noted previously, incompatible compiler versions, the resulting recompilations required, and possible machine-dependent errors are also problems. Unless a private,

unchanging compiler is used, time must be reserved in the development schedule for this type of updating activity.

IX. DISCUSSION

Flexibility was one key factor in the decision to use a high-level language, and it proved to be one of the primary assets of the development technique. Since XPF was written in PL/1, it could be fine-tuned with less effort than if it were written in assembly language. Sections of code could be rewritten in a relatively short period of time. This made it feasible to experiment with implementation methods until optimal code was produced.

One benefit of development in PL/1 that was not considered in the original decision was the ease with which transfer of responsibility is accomplished. Partial turnover of personnel occurred throughout the project. The transfer of code responsibility to new personnel was accomplished very smoothly with no apparent decrease in productivity. Since PL/1 can be largely self-documenting through the use of meaningful variable names and standard operation symbols, it is easy to read and understand. This ease of understanding was the primary reason for the smooth personnel transitions.

Perhaps the most important advantage of developing a system in a high-level language is that the compiler provides area management, storage allocation, error control, data access, and I/O interfaces. The programmers can devote their time to acquiring expertise in the unique requirements of the system.

APPENDIX

Over a period of two years (by Release 8), XPF grew to approximately 32,000 PL/1 plus assembly-language statements. Almost all the

Table I — Comparative productivity, Release 9

	SNX Assembler	CLC Simulator	XPF
Total no. of subroutines	72	90	231
Total no. of source statements	69,788	63,737	34,344
No. of subroutines added or changed	40	30	84
Percent of total subroutines	55.5	37.7	36.3
No. of source statements added or changed	3,286	2,336	7,342
Percent of total source statements	4.7	3.6	21.3
Man-months programmer, management, librarian	23.5	10.5	37.0
Statements per man-month	140	222	198
Man-months, programmers only	18.4	9.0	30.0
Statements per man-month	177	259	244

changes for each release were planned increases in capability, although some, of course, were fixes for bugs. The total effort to produce the first eight releases, debugged and tested, was 222 man-months. The average productivity over this time is therefore about 140 statements per man-month.

Table I compares Release 9 of XPF to the corresponding releases of the SNX assembler and the CLC simulator, both written in assembly language. The simulator was a considerably easier task than XPF for this release because the simulator was only receiving maintenance, while 21.3 percent of XPF was rewritten to add major new capabilities. Nevertheless, the total number of machine instructions produced (per man-month) by the XPF group was greater because they were coding in PL/1, whereas the assembler and simulator groups were coding in assembly language.